Analysis of Sorting Algorithms in C++

This report aims to assess the effectiveness of some commonly used sorting algorithms: Bubble Sort, Quicksort, Mergesort, and Insertion Sort. We examine how they perform on various types of input data, including random, sorted, reverse-sorted, and almost-sorted values. Our objective is to explore how these algorithms function under different circumstances, taking into account aspects such as running time, branch mispredictions, and memory proximity.

Methodology

Implementations:

1) Baseline Implementations:
    - Bubble Sort in C++
    - Standard implementations using C++ STL functions (qsort, std::sort)
2) Optimized Implementations:
    - Quicksort and Mergesort in C++ with manual optimizations for memory access patterns
3) Input Data:
    - Arrays of varying length (1000, 10000, 1000000)
    - Differently sorted (random, sorted, reverse-sorted, almost-sorted)

Results

Running Time:

I chose to present the results for running time with length of array 10000 and 100000, as the results for all the sorting algorithms were indistinguishable for length 1000. These times were calculated using the clock_gettime function in C++

Algorithm	Input Type	Values	Time (ms)
bubble_sort	random sorted	10000	105.13
bubble_sort	sorted sorted	10000	40.8
bubble_sort	reverse sorted	10000	35.33
bubble_sort	almost sorted	10000	41.91
qsort	random sorted	10000	0.72
qsort	sorted sorted	10000	0.18
qsort	reverse sorted	10000	0.22
qsort	almost sorted	10000	0.33
std::sort	random sorted	10000	0.44
std::sort	sorted sorted	10000	0.06
std::sort	reverse sorted	10000	0.05
std::sort	almost sorted	10000	0.2
quicksort	random sorted	10000	0.5
quicksort	sorted sorted	10000	30.97
quicksort	reverse sorted	10000	29.74
quicksort	almost sorted	10000	3.9
mergesort	random sorted	10000	0.89
mergesort	sorted sorted	10000	0.38
mergesort	reverse sorted	10000	0.35
mergesort	almost sorted	10000	0.49
insertion_sort	random sorted	10000	11.86
insertion_sort	sorted sorted	10000	0.01
insertion_sort	reverse sorted	10000	23.58
insertion_sort	almost sorted	10000	0.11
bubble_sort	random sorted	100000	13100.04
bubble_sort	sorted sorted	100000	3587.64
bubble_sort	reverse sorted	100000	3515.66
bubble_sort	almost sorted	100000	3685.22
qsort	random sorted	100000	8.45
qsort	sorted sorted	100000	2.14
qsort	reverse sorted	100000	2.57
qsort	almost sorted	100000	3.48
std::sort	random sorted	100000	5.56
std::sort	sorted sorted	100000	0.78
std::sort	reverse sorted	100000	0.57
std::sort	almost sorted	100000	1.98
quicksort	random sorted	100000	5.96
quicksort	sorted sorted	100000	3129.32
quicksort	reverse sorted	100000	2952.81
quicksort	almost sorted	100000	314.83
mergesort	random sorted	100000	10.76
mergesort	sorted sorted	100000	3.43
mergesort	reverse sorted	100000	3.95
mergesort	almost sorted	100000	5.01
insertion_sort	random sorted	100000	1203.78
insertion_sort	sorted sorted	100000	0.08
insertion_sort	reverse sorted	100000	2452
insertion_sort	almost sorted	100000	0.97

These are some of the insights that can be drawn from the data about running times, provided in the table above:

- std::sort stands out as the quickest sorting algorithm for all input sizes and orders, displaying exceptional speed and scalability when compared to other algorithms.
- qsort is highly effective and nearly as fast as std::sort, albeit slightly less speedy. It performs well with various input sequences but is not as speedy as std::sort.
- Bubble sort's time complexity of $O(n^2)$ causes it to perform poorly, particularly on larger datasets. Significant performance drops are observed on random and reverse sorted data due to high sensitivity to input order.
- Merge sort performs consistently well with different input arrangements and is efficient with larger data sets, albeit not as fast as std::sort.
- The performance of the custom quicksort implementation differs depending on the input order and is typically slower than std::sort and qsort.
- Insertion sort works efficiently on almost sorted data but faces challenges with randomly and reversely sorted data, resulting in slower performance overall when compared to other algorithms.

Conclusively, for most situations, std::sort is the best option for sorting because of its high performance and reliability with different types of input. Merge sort or std::sort are the best choices for situations involving very large datasets. Bubble sort and insertion sort are typically not ideal for sizable or intricate datasets because of their quadratic time complexity.

Branch Mispredictions:

The main contributors to branch mispredictions in our application were on a small number of key functions. The analysis from the perf tool offers valuable insights into the locations where these mispredictions are happening (input array length = 10000):

- The main cause of branch mispredictions is the bubble_sort function with a contribution of about 84.48%. This sizable percentage indicates that the control flow of the function is very susceptible to mispredictions, mostly because of its nested loops and frequent checks of conditions.
- The ‘std::__introsort_loop’ function inside the std::sort function also significantly adds to branch mispredictions as well with a contribution of about 10.05%. Even though its internal sorting mechanism is efficient, it brings about some complexity that impacts branch prediction.
- The custom implementation of quicksort shows a moderate impact on branch mispredictions with about 1.24% of branch mispredictions. Although less significant than bubble_sort, optimizing this function could further enhance performance.

Memory Proximity:

The ‘valgrind --tool=cachegrind’ profiling offers important insights into cache utilization and memory access behaviors (input array length = 10000):

- Instruction Cache (I1): The instruction cache shows a miss rate of 0.00%, indicating efficient instruction fetching with no significant cache contention.
- Level 1 Data Cache (D1): The data cache miss rate is 3.5%, with a higher miss rate for read operations (5.6%) compared to write operations (0.0%). This suggests that read operations may be optimized to reduce cache misses
- Level 2 Cache (LL): The Level 2 cache also shows a 0.0% miss rate, indicating excellent cache utilization and minimal data retrieval delays.

In conclusion, with its poor locality and frequent data swapping, the bubble sort algorithm shows inefficient cache usage resulting in high execution time and significant cache misses. Both std::sort and quicksort exhibit better performance by effectively utilizing cache and reducing execution times with lower cache miss rates. A couple of things could be done if one was looking to maximize performance and efficiency while minimizing cache misses:

- Upgrade or enhance algorithms such as bubble sort in order to improve cache efficiency and speed of execution.
- Pay attention to enhancing data access patterns in order to minimize cache misses, particularly for operations that involve reading data frequently.

Conclusion

This report evaluated the efficiency of Bubble Sort, Quicksort, Mergesort, and Insertion Sort using different data types and measurements. std::sort was found to be the most efficient and adaptable, surpassing other algorithms in all situations, while Bubble Sort fell behind due to its quadratic time complexity. Bubble_sort was the main reason for branch mispredictions, with some added impact from std::__introsort_loop in std::sort. Cache profiling demonstrated that std::sort and quicksort made efficient use of the cache, while Bubble Sort had poor cache utilization. The recommendation is to improve performance by concentrating on enhancing algorithms to be more cache efficient and minimizing branch mispredictions. Additional performance gains could be achieved through further exploration of algorithm improvements and data access patterns.